Command Similarity Measurement Using NLP

Hussain, Zafar

Schloss Dagstuhl - Leibniz-Zentrum für Informatik
2021-08-10

Hussain , Z , Nurminen , J K , Mikkonen , T & Kowiel , M 2021 , Command Similarity Measurement Using NLP . in 10th Symposium on Languages, Applications and Technologies (SLATE 2021) . Open Access Series in Informatics , vol. 94 , Schloss Dagstuhl - Leibniz-Zentrum für Informatik , Dagstuhl, Germany , pp. 13:1 , Symposium on Languages, Applications and Technologies , Portugal , 01/07/2021 . https://doi.org/10.4230/OASIcs.SLATE.2021.13

http://hdl.handle.net/10138/333414
https://doi.org/10.4230/OASIcs.SLATE.2021.13

Command Similarity Measurement Using NLP

Zafar Hussain
Department of Computer Science, University of Helsinki, Finland

Jukka K. Nurminen
Department of Computer Science, University of Helsinki, Finland

Tommi Mikkonen
Department of Computer Science, University of Helsinki, Finland

Marcin Kowiel
F-Secure Corporation, Poland

Abstract

Process invocations happen with almost every activity on a computer. To distinguish user input and potentially malicious activities, we need to better understand program invocations caused by commands. To achieve this, one must understand commands' objectives, possible parameters, and valid syntax. In this work, we collected commands' data by scrapping commands' manual pages, including command description, syntax, and parameters. Then, we measured command similarity using two of these – description and parameters – based on commands' natural language documentation. We used Term Frequency-Inverse Document Frequency (TFIDF) of a word to compare the commands, followed by measuring cosine similarity to find a similarity of commands' description. For parameters, after measuring TFIDF and cosine similarity, the Hungarian method is applied to solve the assignment of different parameters' combinations. Finally, commands are clustered based on their similarity scores. The results show that these methods have efficiently clustered the commands in smaller groups (commands with aliases or close counterparts), and in a bigger group (commands belonging to a larger set of related commands, e.g., *bitsadmin* for Windows and *systemd* for Linux). To validate the clustering results, we applied topic modeling on the commands' data, which confirms that 84% of the Windows commands and 98% ofthe Linux commands are clustered correctly.

2012 ACM Subject Classification Computing methodologies → Natural language processing

Keywords and phrases Natural Language Processing, NLP, Windows Commands, Linux Commands, Textual Similarity, Command Term Frequency, Inverse Document Frequency, TFIDF, Cosine Similarity, Linear Sum Assignment, Command Clustering

Digital Object Identifier 10.4230/OASIcs.SLATE.2021.13

1 Introduction

While users dominantly operate their computers with graphical user interfaces, the operating system converts the mouse clicks to process invocations and other low-level commands. These commands can be stored in textual form to logfiles and resemble the command line (CLI) commands, which users type to command prompts in terminal windows. The stored command history is useful for diagnostics, learning the prevailing commands, user profiling, and various other purposes.

In this work, we aim to help cybersecurity specialists to detect similar commands via textual analysis of commands' man-pages. In particular, we want to understand when two commands are doing the same thing. This is a basic building block of a more complex system to detect anomalous commands. It is important as attackers may try to hide their operation by using unusual commands. To detect any unusual command, it should be compared with other prevalent commands. A user might use alternate commands (aliases) to perform same task. These alternate commands can be valid but because of the different command names,

there are possibilities that they will not be associated with any prevalent command or a group of usual commands. To handle this situation we want to create a system, which will be used as reference, to learn about the unusual command. When an unusual command is encountered, the reference system will be checked, and if there are alternate commands for this unusual command, this command should be marked as safe. One example of this scenario is `del` command. If `del` is encountered as the unusual command, by referring the system we can learn that `del` is similar to `erase` command (which has already been used), so `del` should be marked as safe. In case the new command does not have any similar command, and cannot be found in the command history (logfiles), then the cybersecurity specialists will require further actions. Therefore, we develop command similarity measures, which indicate how strongly two commands resemble each other.

An obvious way to measure command similarity is to compare two command strings with each other. However, it is a poor approach if two commands, such as `del` and `erase`, look very different but do the same thing. Therefore, instead of comparing explicit command strings, we compare command descriptions, available in the documentation manuals. For this, we use natural language processing (NLP) technology. In this way, we develop measures that better detect the semantic similarity of commands than operating only at the command syntax level. Besides studying the command description we also study the command parameter descriptions because they have a major impact on what the command will ultimately do.

The essence of our work is that instead of comparing the raw commands, we compare their documentations. We start from Windows and Linux manual pages, and extract command syntax, description, and parameter information. We then use NLP techniques to analyze the texts. Notice that instead of users explicitly typing the commands, many are generated by a Graphical User Interface (GUI), scheduled tasks, or system activities, which result in new process invocation with its command-line arguments. The key contributions of this paper are (i) showing how NLP techniques are useful in evaluating command similarity; (ii) presenting results of Windows and Linux command similarities with statistics and examples; and (iii) clustering commands based on their similarity.

2 Background and Related Work

A program can be started by another program via operating system API directly or indirectly. A user can invoke a program by interacting with GUI or from a command-line prompt. Programs are also invoked by scheduled tasks and other system activities. Commands have a general form of `param0(command) param1 param2 param3 ... paramN`, where `param0` is typically a name or path of the program to be invoked and `param1` to `paramN` are command parameters (also called flags or switches). Moreover, parameters can be comprised of two or more parts that are logically separated. As the syntax of parameters is not standardized, developers may end up using inconsistent conventions. Concerning command similarity detection, several approaches have been proposed, including (i) measures based on command history, which tries to predict what the user is likely to type next [12, 5]; (ii) converting program fragments to natural language and using NLP for a different kind of processing of these, e.g. for similarity detection of code fragments [19]; and (iii) string similarity comparison either at a character or at token level [6]. There are some tools and methodologies related to the Command-Line parsing. These include NL2Bash[15], a parsing method to map English sentences to Bash commands; ExplainShell[1] to see the help text that matches each argument

[1] `https://explainshell.com/`

of a given command; a documentation browser[2], which finds the relevant docs by parsing the given code and connecting parts of it to their documentation; and a fuzzy hashing method [11], which computes context triggered piecewise hashes(CTPH) to match inputs with sequences of identical bytes in the same order. These programs and methodologies can be used to understand the commands, their explanations, and structure.

However, little research is available in the exact domain of our work. There are papers on the comparison of CLI commands and GUI [21], aiming to find out the strengths/weaknesses of these systems, on the use and importance of CLIs, Adaptive CLIs [4, 8], and Statistical approaches to create profiles from CLIs [9]. The closest work to ours is in the field of linguistics, where the focus is on syntax and semantics of commands. For instance, [14, Chapter 5] provides a linguistic sketch of the Unix language family, which introduces how the Unix commands are written, conjoined, and embedded. However, it does not study the similarity between commands. Another closely related field is code similarity, where interest is in differentiating between representational and behavioral similarities [10]. For instance, text-based approaches seek similarities at characters level and token-based approaches perform token-based filtering and detect renaming of variables, classes, and so on. By defining behavioral similarity with respect to input/output, code fragments can be run to find the similarities, and deep learning can be used to find similarities among code [19]. As discussed earlier, our research goal is to reach a better understanding of the commands' similarities. One way to find similarities among commands is to compare the text by extracting the vector representations and finding the similarities of these vectors to detect similar commands. However, our approach of moving from command to corresponding natural language description and performing comparison at that level has not been studied before.

3 Applying NLP mechanisms

Data Collection

We collected Windows data by web scrapping the official documentation[3] of Windows commands. For web scraping, a Python library Beautiful Soup [17] is used. We selected four parameters for each command, which are the command name, command description, command syntax, and command parameters. We selected a total of 685 commands.

For Linux, no official command documentation was found, but open source web projects for Linux manual pages are available. One of these projects is "The Linux man-pages project"[4], which contains documentation of commands from man pages 1 to man pages 8. We only used man pages 1 and man pages 8, which are for "User commands" and "Superuser and system administration commands" respectively. Four parameters were selected for Linux commands as well, including the command name, its description, synopsis, and parameters. With Beautiful Soup, we collected 2247 Linux commands.

[2] http://showthedocs.com/
[3] https://docs.microsoft.com/en-us/Windows-server/administration/Windows-commands/Windows-commands
[4] https://www.kernel.org/doc/man-pages/

Data Wrangling

Since some of the Windows commands had only the parameter "/?", which could be problematic when comparing with each other, this parameter is excluded from the dataset. The rationale for excluding this parameter is that if two commands with different objectives have only the "?" parameter while comparing them will result in high similarity, which would affect the results. Similarly, "-help", "?", and "version" parameters were also excluded from Linux commands, as there were some commands with only these parameters. All other parameters are kept as they were for further utilization.

Similarity Measures

Several methods are considered to calculate the similarities between commands, such as Word2vec [16] and GloVe [18]. Since Word2vec and GloVe are trained on general English (news, Wikipedia) corpus we did not find embedding useful for the CLI manuals comparison. Since commands data does not have too much variation in the topics, applying any pre-trained model would not be a plausible solution. Generally, pre-trained models work well for document similarities when the topics are of different domains, for example, science and religion, politics and technology, economics and sports, etc. This data is just about commands' description, which with minor variations, contain similar vocabulary and are of the same domain.

TFIDF

A simple yet powerful method to calculate document similarity is to use a document vector based on the Term-Frequency Inverse-Document-Frequency (TFIDF) on the bag of words, with cosine similarity. Generally, in text mining, documents are represented as vectors where the elements in the vectors reflect the frequency of terms in documents [20]. Each word in a document has two types of weights. Local weights are normally expressed as Term Frequencies (TF), and global weights are stated as Inverse Document (IDF) Frequency [20]. TF represents the number of times a word occurred in a document, whereas, IDF gives the weight of a term. Multiplying TF and IDF result in the TFIDF score of a word in a document. The higher the score, the more relevant that word is in that particular document. TFIDF often produces higher scores for words related to the topical signal of a text and lower scores for the words with high frequency, it is well suited for tasks that involve textual similarity.

Cosine Similarity

Cosine Similarity produces a metric, which indicates how related are two documents by looking at the angle between two documents instead of magnitude [7]. Using cosine measure on the top of TFIDF vectors gives a similarity score of two text documents.

Linear Sum Assignment

To compare one command's parameters with another command's, an $n_1 \times n_2$ matrix is created, where n_1 are the total parameters of command 1 and n_2 are the total parameters of command 2. This matrix shows the similarity score of parameters of one command with the parameters of another. Since the objective of this analysis is to find out the similarity score of one command with another based on their parameters, it is required to define a single similarity score based on $n_1 \times n_2$ parameter similarity matrix. To create such matrix, the Hungarian method [13] is used. The method solves the Linear Sum Assignment

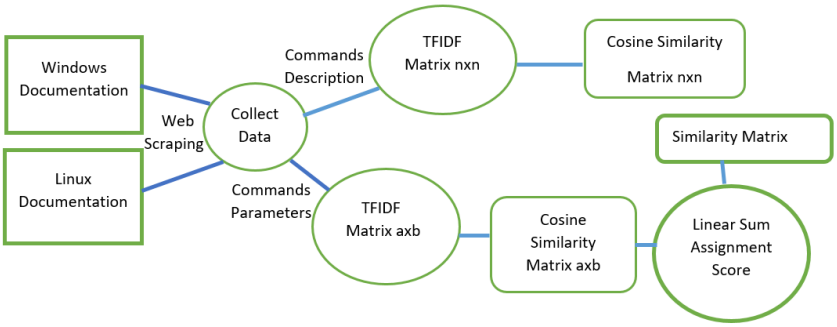


Figure 1 A Step-wise Overview of the Mechanism.

Problem (LSAP), one of the most famous problems in linear programming and combinatorial optimization [3] Given an $n \times n$ cost matrix C, each $C_{i,j}$ is the cost of matching vertex i of the first partite set and vertex j of the second set [3]. The objective is to match each row to a different column so that the sum of the corresponding entries is minimized. In other words, we select n elements of C so that there is exactly one element in each row and one in each column and the sum of the corresponding costs is minimum [3]. The formulation of this algorithm is $min\Sigma_i\Sigma_j C_{i,j}X_{i,j}$, where X is a boolean matrix with $X_{i,j}$=1 iff row i is assigned to column j. Tuning this formula, as shown below, by multiplying the similarity matrix with -1 and normalizing it with the minimum number of parameters among two commands, returns a value that indicates how similar two commands are based on their parameters.

$$\frac{-min\Sigma_i\Sigma_j C_{i,j}X_{i,j}}{min(n_1, n_2)} \tag{1}$$

Command similarities based on the subset of data

As the data contains descriptions and parameters, we use both as a basis for comparison. The rationale for two different comparisons is to analyze which commands have similar objectives. We believe that comparing descriptions of programs results a group of programs performing similar actions, like data compression, file manipulation, or network tools. Another point of interest is comparing commands parameters, to consider which of the commands have common flags.

Similarities based on command description. First, commands are compared based on their description. A TFIDF matrix is generated after removing the stopwords, or frequently used words that do not carry any thematic meaning, like the, is, at, or an. After creating the TFIDF matrix, cosine similarity is calculated for all the commands. The generated $n \times n$ matrix, with diagonal values as 1, gives cosine similarities of each commands' description with other commands.

Similarities based on command parameters. For the comparison of the parameters, first, the TFIDF matrix of commands is generated after removing the stopwords. This matrix is of $n_1 \times n_2$ shape, where n_1 are the total number of parameters of command 1 and n_2 are the parameters of command 2. To calculate a single similarity score, the LSAP method is applied on top of this TFIDF matrix. This resulted in an $n \times n$ matrix where n is the total number of commands.

Similarities based on command description and parameters combined. Finally, for each command, parameters' text is appended at the end of description text but the parameters' names are excluded from this textual analysis. Following the same procedure, the TFIDF matrix is created for all n commands. Using the matrix, the cosine similarity of the commands' overall text is calculated. This resulted in an $n \times n$ matrix, which shows the commands' overall similarity.

4 Results

After creating the $n \times n$ matrix of command similarities, Windows and Linux commands are visualized in the separate graphs. As discussed above, these similarities are based on description, parameters, and the overall (description + parameters) commands textual data. The starting point of our analysis is to find out the ratios of commands with high similarities and low similarities. Commands with high similarities are those that have at least one other command with a similarity score of 0.75 or more. The rationale for selecting 0.75 as a threshold is that it will give us commands which have 3/4th of similar text, which in general is a good enough threshold of comparing two texts. For example in Windows commands, based on the commands' description, `md` has highest similarity score (equal to 1) with `mkdir`, whereas `sxstrace` have highest similarity with `pathping` with a similarity score 0.088. So, `md` falls under the high similarity tag (sim >=0.75) and `sxstrace` falls under the low similarity tag (sim <0.75).

One reason for the commands falling under the low similarity tag is the very short description. It is also worth mentioning that changing the threshold values, will affect the results. The result shows that commands that fall under the low similarity tag can be assumed as the isolated commands, as they are not related to other commands and generally they have no alternative commands either. In contrast, commands that fall under the high similarity tag are the ones either from the same domain such as windows' `bitsadmin*` or have the alternative commands such as `del` and `erase`.

When compared based on their descriptions, most commands fall in the low ranges (sim less than 0.75). These commands do not show high similarity with other commands. These commands need more attention when working with them, as any analysis for the commands with high similarity such as `md` and `mkdir` can be easily applied and interpreted, as they have a similarity score of 1 based on their description. For example, a program supposed to learn the behavior of `md`, will easily learn the behavior of `mkdir` also.

Table 1 shows that Windows commands, when compared based on their description only, have a small percentage of high similarity commands. 79.1% of the commands are in the low similarity ranges, which indicates that based on the description, Windows commands are spaced out from each other. But when these commands are compared based on their parameters, there are almost 41% of the commands with high similarity. On verifying this result, it appears that some commands are from a bigger group which have similar parameters set with each other, for example, `manage bde*` and `logman*`. When Windows commands' descriptions are extended with parameters, the result shows that 72% of the commands fall in the low similarity ranges and a mere 28% of the commands have one or more similar commands.

Comparing Linux command similarity percentage shows almost the same results as in Windows commands similarities for description and overall. Around 74% of the commands based on their descriptions fall in the low similarity ranges, whereas parameters similarities show that 31.5% of the commands have similar parameters. If we compare parameters

Table 1 Percentage of Windows and Linux Commands in Different Similarity Ranges.

Sim. ranges	Descr.	Params	Overall
Sim >=0.75 Windows	20.9%	40.7%	27.9%
Sim >=0.75 Linux	26.3%	31.5%	29.1%
Sim <0.75 Windows	79.1%	59.3%	72.1%
Sim <0.75 Linux	73.7%	68.5%	70.9%

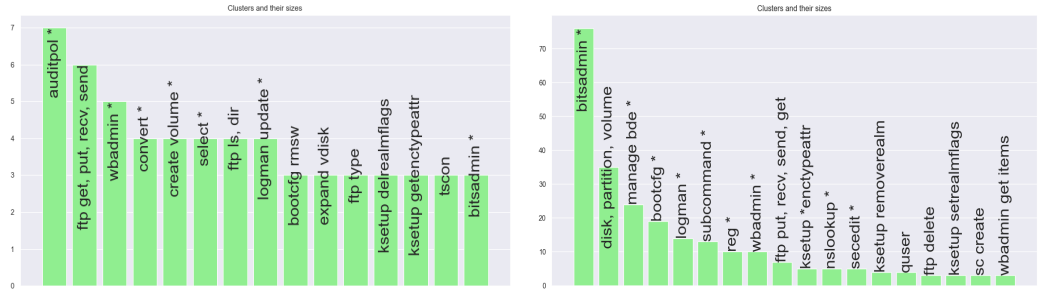
similarities result of Windows and Linux, there is a difference of almost 10%. One reason for this difference could be a high number of parameters for Linux commands which separate the commands quite well. Just like Windows commands, there are some groups of Linux commands also, such as `lttng*`, `pmda*`, and `systemd*`, which share almost the same parameters within their group, but other than these groups, parameters are quite different from each other. Overall, Linux commands show that 29% of them have one or more similar commands, whereas almost 71% of them fall in the low similarity ranges.

4.1 Windows Commands

As discussed earlier, Windows commands are compared based on the commands' description, parameters, and overall. Here we are sharing only the clusters of highly similar commands (in which we are interested) based on these three scenarios. The clusters are created by selecting a command and comparing their similarity score with other commands. If any other command has a similarity score of 0.75 or more with the selected command, an edge is created between the two commands. Applying this method for all the commands results in a graph where commands form communities (for the consistent connotations, we call them clusters). Some commands form a bigger cluster as it has already been discussed and some create a cluster of just two or three commands. Since showing hundreds of clustered commands in one graph reduces the readability, we selected the method of showing the count of commands in each cluster. Figure 2a shows the clusters of commands with high similarities based on the commands' descriptions. The clusters with at least three commands are selected to show the group of commands which share a similar description. The first and the biggest cluster in this analysis is made of commands such as `auditpool backup`, `auditpool get`, `auditpool restore` etc., whereas the second cluster is made of `ftp get`, `ftp send`, `ftp recv` etc. The interesting factor is to note the cluster seven which contains the commands such as `ftp ls`, `ftp dir`. This cluster is separated from cluster 2 which, though contains `ftp` commands, are doing a different job as compared to the commands in cluster six. This verifies that a simple but powerful approach, such as TFIDF along with Cosine Similarity measure, can successfully separate the commands based on their description.

Figure 2b shows the clusters of commands with high similarities based on the commands' parameters. The biggest cluster which has 76 commands in it, is of the `bistadmin*` commands which are used to create, download or upload jobs, and to monitor their progress. On manually verifying the Windows' manual pages, it proves that the `bitsadmin*` commands share the same parameters. The second cluster contains the commands such as `create partition`, `attributes disk`, `online volume` etc. The next clusters are made of commands of their groups, such as `manage bde*` makes a cluster of their own, whereas `bootcfg*` creates a separate cluster. Similarly, `logman*` commands have a different cluster, whereas `reg*` commands fall in a separate cluster.

The third scenario is the extension of the commands' description with their parameters. The clusters with at least 3 commands are selected only to see the groups of commands. Figure 3 shows that the biggest cluster is of `bitsamdin*` commands. By comparing the



(a) Description Similarities. **(b)** Parameters Similarities.

Figure 2 Windows commands clusters.

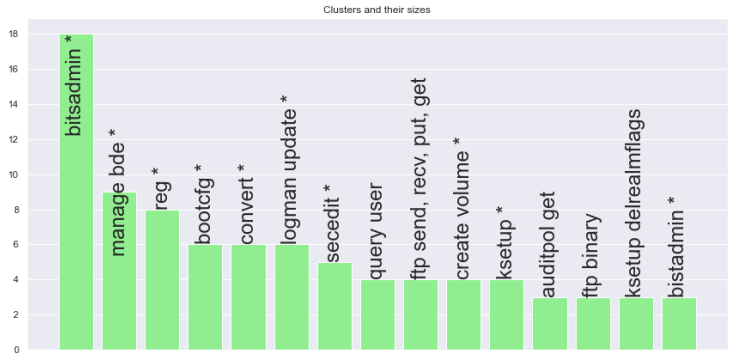
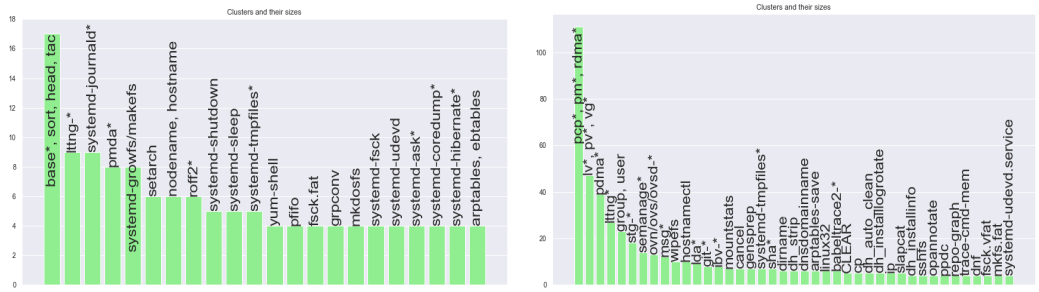


Figure 3 Windows commands clusters overall similarities.

size of this cluster with the Figure 2b which is of windows' parameters similarities, it is worth noticing that comparing the `bitsamdin*` commands' complete textual-data, reduces the cluster size. A total of 76 `bitsamdin*` commands are highly similar when compared based on their parameters, but when we compare the complete text, the size of the cluster reduces to 18. This indicates that overall 18 `bitsamdin*` commands are similar to each other, but 76 `bitsamdin*` commands share the same parameters. These separate analyses help us distinguish the commands sharing the same parameters, the commands sharing the same description, and the similar commands overall. The rest of the clusters are formed by `manage bde*` commands, `reg*` commands, `bootcfg*` commands and so on.

4.2 Linux Commands

Following the same processes of calculating commands' similarities, Linux commands are compared. In the first scenario, commands' description are compared as shown in the Figure 4a. Only the clusters with at least 4 commands are considered for Linux analysis. The biggest cluster in this scenario is based on 17 different commands, such as `base32`, `base64`, `head`, `expand`, `sort`, `shuf` etc, which are sharing the same description. The `lttng*` commands such as `lttng-save`, `lttng-load`, `lttng-start`, `lttng-stop` etc. form a cluster of their own, whereas `pmda*` commands such as `pmdakernel`, `pmdalinux` etc. constitute a separate cluster. The interesting point in this scenario is the `systemd*` commands which are forming multiple clusters based on their jobs specifications. In the second scenario, commands' parameters are studied and their similarities are calculated.



(a) Description Similarities. **(b)** Parameters similarities.

Figure 4 Linux commands clusters.

The commands with highly similar parameters are clustered together as shown in Figure 4b. A total of 111 commands which are from different groups, but share the same parameters, are making the biggest cluster in this scenario. The commands such as `pmlogsummary`, `pcp-vmstat`, `pmevent`, `pmdumplog`, and `pmstat` are examples of this cluster. A point worth mentioning here is that all 111 commands in this cluster do not necessarily need to be connected, as one command in this cluster can have a similarity of exact 1 with another command and at the same time a similarity of 0.8 with a third command. Now the second and third command can be disconnected from each other but they all are in the same cluster. As it was in Windows commands' parameters similarities, here also commands from the same group are making clusters of their own such as `stg*`, `msg*`, and `git*` commands.

The final scenario is the Linux commands' overall comparison. Figure 5 shows that the biggest cluster consists of 15 commands. It includes commands `lvmdiskscan`, `lvscan`, `pvresize`, `pvchange`,and `vgmerge` etc. These are all from Linux Superuser and system administration commands (i.e. man pages 8). The second cluster contains the commands such as `pcp2csv`, `pcp2json`, and `pcp2xml` etc. Similarly, `lda*` commands, `pmda*` commands, `git*` commands, and `yum*` are all in their own separate clusters. These results show that the combination of TFIDF and Cosine Similarity can separate the text quite well.

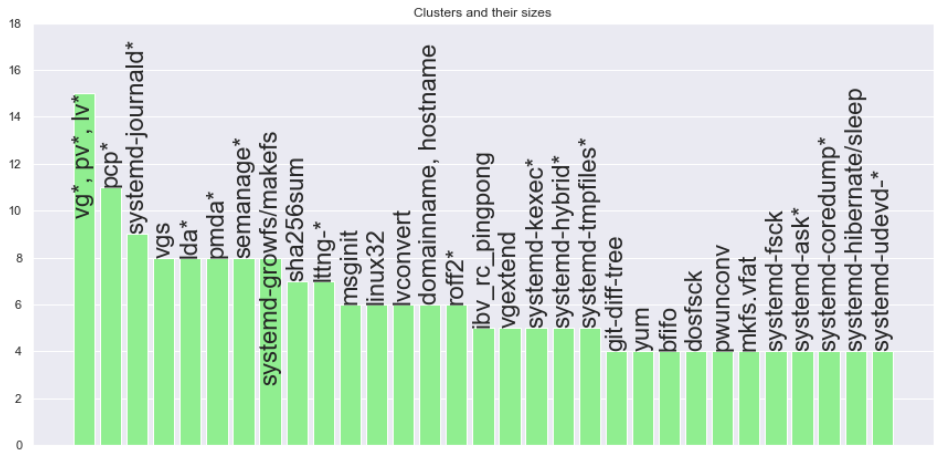


Figure 5 Linux commands clusters overall similarity.

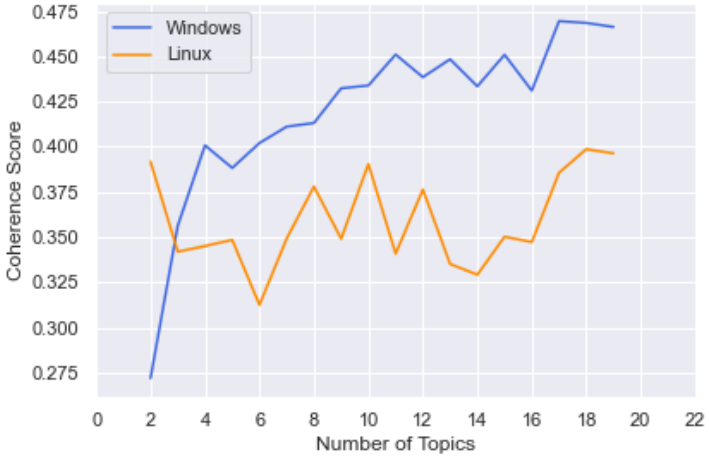


Figure 6 Number of Topics vs Coherence Score.

These results show that most of the Windows and Linux commands are not easily separable from their description as more than 60% of the commands fall between the similarity score of 0.25 and 0.75. The commands at the extreme ends are easily separable as either they have highly similar commands or they are unique. When commands are compared based on their parameters, the middle range ratios decreased for both Windows and Linux commands, indicating that commands (or group of commands) are more easily distinguishable from each other based on their parameters.

4.3 Clusters Validation With Topic Modeling

A topic model is a type of statistical model for discovering the abstract "topics" that frequently occur together in a collection of documents [1, 2]. Topic modeling is a frequently used text-mining tool for the discovery of hidden semantic structures in a text body [2].

To validate these clustering results, we applied topic modeling on Windows and Linux commands' descriptions, separately. We executed the model for various number of topics, starting from 2 up to 20. To find the optimal number of topics, we compared the coherence score against the specified number of topics. Figure 6 shows the number of topics on x-axis and coherence score on y-axis. The result indicates that the best coherence score for Windows and Linux can be achieved by setting the number of topics to 18. The topic modeling for Linux shows that the best coherence score can be achieved by setting the number of topic as 4 or 5, but with 2247 Linux commands, setting a low value as the number of topics, will not give us any useful insights. The coherence score is 0.465 for Windows OS commands, and 0.408 for Linux OS commands. Theoretically, coherence score of close to 1 is considered as optimal, but depending on the problem in hand and available textual data, optimal coherence score varies. In our case, we have a very specific vocabulary, which indicates the low diversity in the vocabulary. With this vocabulary, a coherence score of more than 0.4, is considered good, which is also verified in the Table 2, where four different commands (which belong to the same family) are given the same topic. Therefore, we used 18 as the optimal number of topics for both Windows and Linux commands.

For validation, we selected those commands which have one or more similar commands based on their descriptions. We passed each of these command's description to the model to get a topic number. For example, in Figure 2a, four `create volume *` commands are

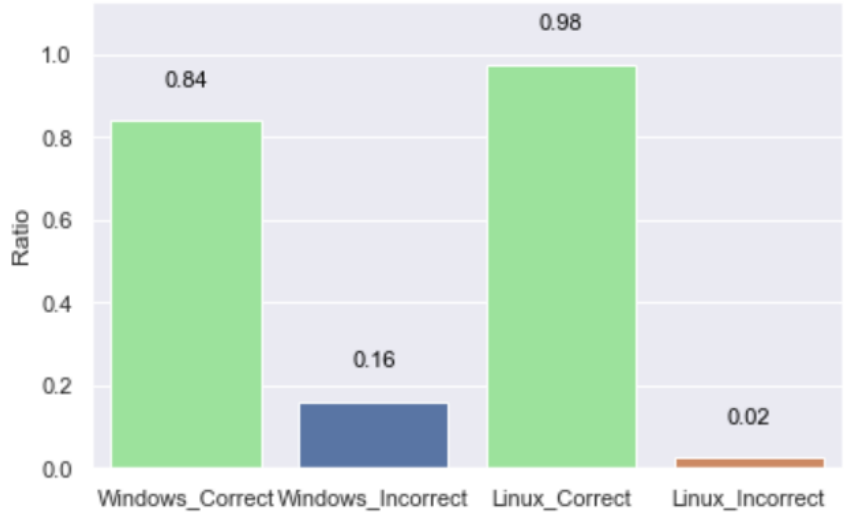


Figure 7 Ratio of Windows & Linux Commands Clusters Validated by Topic Modeling.

making a cluster of their own. We passed the description of these commands to our model, and the model suggested that these commands belong to Topic 1. Table 2 shows `create volume *` commands' description and the topic number our model is suggesting. By looking at the topics under Topic 1, we found the following terms: "`disk`", "`volume`", "`path`", "`dynamic`", "`load`", "`metadatum`", "`writer`", "`restore`", "`datum`", "`save`". Looking at the `create volume *` commands descriptions and the topics under Topic 1, we can say that our model is doing a fairly good job.

Table 2 Validating Topic Modeling for Windows Commands (create volume *).

Command	Description	Topic
create volume simple	Creates a simple volume on the specified dynamic disk. After you create the volume, the focus automatically shifts to the new volume.	1
create volume raid	Creates a RAID-5 volume using three or more specified dynamic disks. After you create the volume, the focus automatically shifts to the new volume.	1
create volume stripe	Creates a striped volume using two or more specified dynamic disks. After you create the volume, the focus automatically shifts to the new volume.	1
create volume mirror	Creates a volume mirror by using the two specified dynamic disks. After the volume has been created, the focus automatically shifts to the new volume.	1

Similarly, we validated clustering results of Windows and Linux commands based on their descriptions. Figure 7 shows that 84% of Windows commands and 98% of Linux commands which were in same clusters based on their description also share the same topics. The reason Linux commands show such a high ratio is because there are more than one variant of same command found in the documentation, like `clear` and `CLEAR` commands. We wanted to keep them in their original form as they both are used commonly. These results validate the clusters we created with TFIDF and cosine similarity.

4.4 Testing of Command Similarities

Once the similarity results are validated by the topic modeling, we created an ad-hoc scenario where we receive commands and compare them explicitly. For each pair of commands, if the command names are not same, we look in the $n \times n$ matrix of similarities we created in Section 3. A threshold value of 0.8 is specified, and if two different commands have a similarity score of more than 0.8, we classify them as "Similar". Table 3 shows seven examples of Windows commands, where *Command_1* and *Command_2* are the two commands with different names, *Score* is the similarity score of two commands, and *Result* indicates whether two commands are similar or not. The names of the commands are bold in the table. It can be seen from the table that "query user" and "quser" are the different commands but performing the same task. Their similarity score is 0.989 which indicates that these two commands are aliases and a user can use them alternatively. Similarly, "ftp put" and "ftp send" are performing the same task. By looking in the similarity matrix, we found that their textual similarity is 0.9, which indicates that these two commands are also aliases and can be used alternatively. These are just seven examples, but there are hundreds of Windows and Linux commands which are aliases and should be considered as same commands when comparing them. Without the similarity matrix we created, any system measuring the similarities of commands will either classify the commands and their aliases as "Not-Similar" because of different commands names, or the system will need to be fed manually with all the commands and their aliases, which does not seem an optimal way of doing it. Official documentation[5] of Windows commands can be referred to manually confirm the commands (shown in the Table 3) and their textual descriptions.

Table 3 Testing Windows Commands' Similarities.

Command_1	Command_2	Score	Result
C:\>**query user** user /server:Server64	C:\>**quser** user /server:Server64	0.989	Similar
cd Windows	**chdir** Windows	0.85	Similar
ftp> send d:\web2\script.py	**ftp> put** d:\web1\script.txt	0.9	Similar
del C:\examples\MyFile.txt	**erase** C:\examples\MyFile.txt	0.88	Similar
mkdir C:\test\temp	**md** C:\test \data	0.99	Similar
change port com12=com1	**chgport** com12=com1	0.884	Similar
ftp>get remote_file.py [local_file.py]	**ftp>recv** remote_file.py [local_file.py]	0.882	Similar

5 Discussion

Most computer vulnerabilities can be exploited in a variety of ways. Attackers may use a specific exploit, a misconfiguration in one of the system components, or a secret pathway to gain entry to a computer system. In most of these attacks, hackers run malicious programs through command line commands. One way to detect a hacker or malicious activities on a machine is by looking at sequences of program invocations and their parameters. The detection can be based on a combination of many different methods, from rule engines, the prevalence of the program's invocation, reputation scores, to more advanced machine learning methods. The threat hunters focus on two tasks: first to detect all hacking or malicious activities as quickly as possible and second reduce the number of false positives to a bare minimum.

[5] `https://docs.microsoft.com/en-us/Windows-server/administration/Windows-commands/Windows-commands`

This research can help both of the mentioned scenarios. First of all the web-scraped program manuals will provide a good context database for common programs invocations on Linux or Windows operating system. Secondly, we were able to find program aliases and cluster program families, with the use of a simple and robust method. One could calculate the prevalence of the program clusters instead of individual programs to limit down the number of false-positive detection. Furthermore, the obtained clusters can be used as a validation benchmark for more sophisticated, unsupervised methods, that try to find similar programs based on program command parameters only. Defining similar programs manually would be a very tedious task due to the large number of common programs mentioned in the manuals, and an even bigger set of proprietary software used in the wild.

By enriching the program invocation commands with external sources of data not only provides us the necessary context of the programs but also helps us in understanding the user's intentions and behavior, which is an important factor to detect malicious activities.

This research work resulted in creating a reference database of the Windows and Linux commands. Primarily, this reference database will be used by cyber security specialists to learn about the commands and their aliases, but it can be used for any other use case or research work where the objective is to find prevalent commands and learn the common parameters among them.

6 Conclusion

This research work studies Windows and Linux commands. It helps us in finding the clusters of commands (small and larger groups), the ratios of commands which are isolated or highly similar to other commands, and to reach a better understanding of command similarity than what is possible by simply comparing the command strings as such. These results can be the basic building block of many applications and machine learning models in the cybersecurity domain. These results can also be useful for user profiling, predicting a set of parameters for different commands, and learning the commands and their set of parameters that invoke different programs.

References

1 Rubayyi Alghamdi and Khalid Alfalqi. A survey of topic modeling in text mining. *International Journal of Advanced Computer Science and Applications*, 6, January 2015. doi:10.14569/IJACSA.2015.060121.

2 David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3(null):993–1022, 2003.

3 Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, 2012.

4 Brian D. Davison and H. Hirsh. Toward an adaptive command line interface. In *HCI*, 1997.

5 Brian D. Davison and H. Hirsh. Predicting sequences of user actions. In *AAAI/ICML 1998 Workshop on Predicting the Future: AI Approaches to Time-Series Analysis*, 1998.

6 Najlah Gali, Radu Mariescu-Istodor, Damien Hostettler, and Pasi Fränti. Framework for syntactic string similarity measures. *Expert Systems with Applications*, 129:169–185, 2019. doi:10.1016/j.eswa.2019.03.048.

7 Jiawei Han, Micheline Kamber, and Jian Pei. 2 - getting to know your data. In *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 39–82. Elsevier, third edition edition, 2012.

8 Haym Hirsh and Brian Davison. Adaptive unix command-line assistant. *Proceedings of the International Conference on Autonomous Agents*, October 1998. doi:10.1145/267658.267827.

9 José Iglesias, Agapito Ledezma Espino, and Araceli Sanchis de Miguel. Creating user profiles from a command-line interface: A statistical approach. In *International Conference on User Modeling, Adaptation, and Personalization*, volume 5535, pages 90–101. Springer, 2009.

10 E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy paste. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 78–87, 2010. doi:10.1109/CSMR.2010.33.

11 Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06). doi:10.1016/j.diin.2006.06.015.

12 Benjamin Korvemaker and Russ Greiner. Predicting unix command lines: Adjusting to user patterns. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, page 230–235, 2000.

13 H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. doi:10.1002/nav.3800020109.

14 J. Lawler and H.A. Dry. *Using Computers in Linguistics: A Practical Guide*. Routledge, 1998.

15 Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation LREC 2018, Miyazaki (Japan), 7-12 May, 2018.*, 2018.

16 Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, 2013. arXiv:1310.4546.

17 Vineeth G Nair. *Getting Started with Beautiful Soup*. Packt Publishing Ltd, 2014.

18 Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL: http://www.aclweb.org/anthology/D14-1162.

19 M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 542–553, 2018.

20 M. Umadevi. Document comparison based on tf-idf metric. In *International Research Journal of Engineering and Technology (IRJET)*, volume 7(02), 2020.

21 Antony Unwin and Hofmann Heike. Gui and command-line - conflict or synergy? In *Proceedings of the 31st Symposium on the Interface: models, predictions, and computing*, 2000.